Using Threat Analysis Techniques to Guide Formal Verification: A Case Study of Cooperative Awareness Messages*

Marie Farrell[1], Matthew Bradbury[2], Michael Fisher[1], Louise A. Dennis[1], Clare Dixon[1], Hu Yuan[2], and Carsten Maple[2]

[1] Department of Computer Science, University of Liverpool, UK
[2] Cyber Security Centre, WMG, University of Warwick, UK

Abstract. Autonomous robotic systems such as Connected and Autonomous Vehicle (CAV) systems are both safety- and security-critical, since a breach in system security may impact safety. Generally, safety and security concerns for such systems are treated separately during the development process. In this paper, we consider an algorithm for sending Cooperative Awareness Messages (CAMs) between vehicles in a CAV system and the use of CAMs in preventing vehicle collisions. We employ threat analysis techniques that are commonly used in the cyber security domain to guide our formal verification. This allows us to focus our formal methods on those security properties that are particularly important and to consider both safety and security in tandem. Our analysis centres on identifying STRIDE security properties and we illustrate how these can be formalised, and subsequently verified, using a combination of formal tools for distinct aspects, namely Promela/SPIN and Dafny.

1 Introduction

Emerging applications of autonomous robotic systems include Connected and Autonomous Vehicle (CAV) systems where self-driving vehicles communicate with each other in order to safely travel between different locations. This communication typically occurs over a wireless network that is vulnerable to attacks and these attacks could potentially impede the safety of the passengers. Therefore, ensuring that both cyber security and safety issues are properly addressed during the software development process is crucial for these CAV systems. While a recent survey on formal verification techniques for autonomous robotic systems identified a number of challenges for applying formal methods to these systems [19], cyber security as a distinct challenge for formal methods has often been overlooked. In particular, identifying which cyber security properties to verify is often difficult for formal methods practitioners.

* This work is supported by grant EP/R026092 (FAIR-SPACE Hub) through UKRI under the Industry Strategic Challenge Fund (ISCF) for Robotics and AI Hubs in Extreme and Hazardous Environments.

In this paper, we present a simple case study that employs (informal) threat analysis techniques from the cyber security domain to guide our verification effort of the Cooperative Awareness Message (CAM) protocol, used in vehicle-to-vehicle communications [26]. CAMs are heartbeat messages that are periodically broadcast by each vehicle to its neighbours to provide basic vehicle status information including position, velocity, acceleration, heading, etc. [26]. Since these vehicles communicate over an unsecured network, ensuring that CAMs are secure is crucial as we move toward driverless cars. This is also relevant in other areas where autonomous vehicles communicate with each other, such as a group of vehicles in orbit or rovers mapping an unknown and hazardous environment.

To this end, we contribute a basic methodology for security-minded formal verification and a case study demonstrating our approach using existing formal methods and STRIDE threat analysis. The threat analysis will identify threats that fall into specific aspects of the six categories specified by STRIDE. Since CAMs are for use in CAV systems, which are inherently cyber-physical, we see this case study as an experiment on how to combine threat analysis and formal verification and our approach could be used in the development of other cyber-physical systems.

This paper is structured as follows. In the remainder of Section 1 we outline our basic methodology for security-guided formal verification. Then, Section 2 describes the relevant background material and related work. In Section 3, we present our threat analysis of the CAM protocol using the STRIDE classification. In Section 4, we present our results of analysing how a spoofing attack can impact the safety of a simple, three vehicle CAV system by devising an abstract system model in Promela and using the SPIN model-checker for verification. Section 5 presents a Dafny implementation of the CAM protocol and illustrates how we can verify properties related to Denial of Service and Repudiation. Finally, we conclude and outline future work in Section 6.

1.1 Methodology

In order to enhance the software engineering process and encourage collaboration between cyber security and formal methods practitioners we followed the high-level methodology outlined in Fig. 1. We started by analysing the available documentation that informally describes the CAM messaging protocol [1]. Then we independently carried out threat analysis and construction of formal models of the CAM protocol.

In particular, we constructed two formal models of the protocol. The first is a high-level system model that is written in Promela and verified using the SPIN model-checker which we use to investigate a spoofing attack. The second is an algorithm-level model of the CAM protocol, written in Dafny, which we use to analyse properties related to denial of service and non-repudiation. Finally, we defined formal properties, based on the threat analysis that was carried out, that we then encoded and verified with respect to our formal models of the CAM protocol.

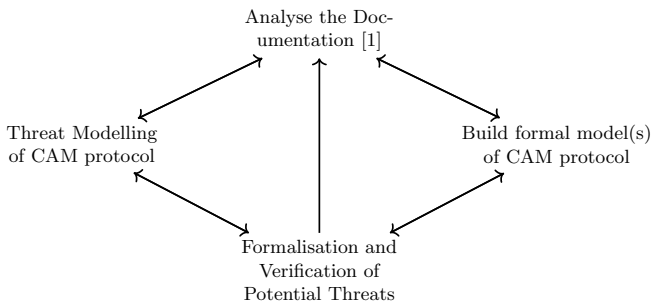

Fig. 1: Our high-level methodology for security-guided formal verification of the CAM protocol involved independently carrying out threat analysis and formal modelling based on the available documentation. Then we used the identified threats to guide our formal verification effort.

In the event that errors or discrepancies were found in either the threat analysis or formal verification, we returned to the documentation for clarification. This is indicated by the arrows in Fig. 1 and allowed us to discern whether the errors were in our modelling of the protocol and/or if the correct level of abstraction was captured by our formal models. By following this methodology we were able to foster cooperation and combine expertise from both the cyber security and formal methods domains. This kind of collaboration can be easily integrated into the development process and can potentially save time since formal verification and security analysis can be combined and used in a complementary fashion.

In practice, the system might be modified as mechanisms are put in place to prevent attacks that were exposed by threat analysis. When this occurs, developers should revise and reverify their models in light of this new behaviour.

2 Background and Related Work

In this section, we present the relevant background material under three distinct headings. First, we describe the threat analysis techniques that we have used, then we discuss formal verification and introduce the tools and techniques that we use throughout the remainder of this paper. We also provide concrete details about Cooperative Awareness Messages (CAMs) as contained in the standard documentation [1]. Finally, we briefly describe related work in this area.

Threat Analysis: When engineering security-critical systems, developers often employ threat analysis techniques to help to identify security vulnerabilities so that targeted mitigations can be put in place. One such technique is the STRIDE [15] classification. In fact, there are many other techniques for threat

modelling, and our approach works equally well with any of them (e.g. CIA which stands for Confidentiality, Integrity and Availability [31]), but for ease of explanation, we adopt STRIDE. This includes: (i) Spoofing - attacker pretends to be another system entity, (ii) Tampering - attacker manipulates data maliciously, (iii) Repudiation - attacker can deny sending a message that it sent, (iv) Information Disclosure - attacker can cause the system to reveal information to those it is not intended for, (v) Denial of Service - attacker can prevent the system from functioning, and (vi) Escalation of Privilege - attacker can perform more actions than allowed. Analysing a system in light of STRIDE threats helps developers to secure the system by identifying vulnerable areas so that mitigations can be included. The identified threats will also have their impact and likelihood assessed in order to calculate the risk of each threat [24], allowing the prioritisation of developing mitigations for threats with a higher risk.

Formal Verification: In order to assure the correctness of a software system, formal methods provide an array of mathematically-based tools and techniques for proving properties about a system. Formal methods are predominantly used in safety-critical systems where a software failure can potentially cause harm. In this paper, we employ two distinct formal methods; Promela/SPIN [10] and Dafny [17] to verify properties about the CAM protocol. In each case, we model the CAM protocol at a different abstraction level; Promela for system-level modelling and Dafny for algorithm-level verification. Since these systems are typically very complex, the use of multiple formal methods is necessary [8], and cyber security threat analysis techniques help us to highlight the most relevant security properties.

Promela is a general purpose programming language, particularly developed for protocol verification, while the patterns of temporal behaviour that can be verified can be complex and varied [9]. SPIN is a model-checker that automatically checks temporal properties over system models encoded in the Promela programming language [9, 10]. Essentially, SPIN explores all possible runs of Promela input models and assesses these against an automaton capturing temporal behaviour that should *never* occur. If all runs have been explored without finding a violation of the temporal properties then the model is valid. If a violation is found, it is returned as a counter-example[3].

Dafny is a programming language that facilitates the use of specification constructs that allow the user to specify pre- and post-conditions as well as loop invariants and variants [17]. Dafny is used in the static verification of the functional correctness of programs. Dafny programs are translated into the Boogie intermediate verification language [3] and then the Z3 automated theorem prover discharges the associated proof obligations [7]. We chose Dafny for this case study because of its similarity to other programming languages making it easy to communicate the verified solution to security engineers that are unfamiliar with formal methods[4].

[3] We used version 6.4.6 of SPIN.
[4] We used version 2.2.0 of Dafny.

Cooperative Awareness Messages (CAMs): As outlined briefly in Section 1, CAM are heartbeat messages that are sent between vehicles in a CAV system. The CAM standard documentation is contained in [1] and we briefly summarise this here in order to give the reader an understanding of the nature of the CAM protocol. In autonomous vehicles, the CA Basic Service is a facilities layer that is responsible for operating the CAM protocol which is composed of two services: (1) the sending of CAMs, including their generation and transmission, and, (2) the receiving of CAMs and the modification of the receiving vehicle's state in light of the received messages. The CA Basic Service is in control of how frequently CAMs are sent and it interfaces to a number of other services, such as the SF-SAP which provides a number of basic security services (including digital signatures and certificates) for CAM [1, §5.1].

CAMs are sent in plain text as they are intended for all vehicles within range of the sender. This also means that time expensive encryption and decryption is not required. However, to ensure the authenticity of the sender (that a message sent from vehicle v actually came from vehicle v), digital signatures are used as they allow a receiver to use the contents of the message, the signature and the public key of the sender to verify its origin. Note that in this paper we are primarily concerned with the protocol for sending and receiving CAMs and the threats that can be identified at this level rather than detailed cryptographic protocols and digital signing.

Once CAMs are received by surrounding vehicles, the receivers can modify their own state based on the received messages. In particular, if a vehicle receives a message from one proceeding it which indicates that the leading vehicle is slowing down, then the vehicle that received this CAM should also slow down in order to avoid collision.

Related Work: iUML-B and refinement in Event-B have been used to analyse a known security flaw called double tagging in a network protocol [30]. Other related work includes the use of the Tamarin prover to formally analyse and identify one known functional correctness flaw and one unknown authentication flaw for a revocation protocol [33]. Here, the revocation is of malicious or misbehaving vehicles from a vehicular networking system. Our work differs to these in that we use threat analysis to guide our verification rather than use formal methods to identify previously known bugs.

Vanspauwen and Jacobs have devised an approach to the static verification of cryptographic protocol implementations using their symbolic model of cryptography formalised in VeriFast [32]. They attach contracts to the primitives in an existing cryptography library. Their focus is on the verification of cryptographic protocols whereas we focus on using cyber security techniques to guide verification rather than verifying cryptographic protocol implementations.

Huang and Kang [11] use a probabilistic extension of the Clock constraint specification language (Ccsl) to analyse safety and security properties related to timing constraints for a cooperative automotive system. They specified a number of safety constraints as well as a number of security constraints including spoofing, secrecy, tampering and availability. Their work facilitates the verifica-

tion, using the UPPAAL model-checker, of safety and security properties related to timing constraints. It does not, however, integrate results from a security engineering perspective in order to define these properties and only focuses on those properties related to timing.

Other related work includes the use of the CSP process algebra for protocol verification [27–29]. Their focus is on authentication [28] and non-repudiation protocols [27]. Their approach involves specifying the relevant protocol, agents and environment in CSP [29]. Notably, they remark that, by modelling the protocol in CSP, they could provide a formal and verified specification of the protocol which allowed them to clarify the, usually, informal protocol description.

Kamali et al. [14] have used formal verification of an autonomous vehicle platooning system to demonstrate the use of different formal techniques for distinct system subcomponents. In this case, autonomous decision-making, real-time properties and spatial aspects. Our approach, presented in this paper uses different formal methods to verify distinct security-related properties of the CAM protocol at different levels of abstraction.

3 Threat Analysis of CAM

In this section, we describe our threat analysis of the CAM protocol. Threat analysis is important for ensuring the security of a system since it is used to identify all of the potential threats to the system. There are a variety of different threat modelling methods including STRIDE, SAHARA, HARA, TARA and others that are suggested in multiple industry standards (i.e. ISO26262, SAE J3061). In this paper, we use the STRIDE classification outlined earlier [15].

3.1 Specialising STRIDE for CAM

CAMs (formatted using ASN.1 as specified in [1, Annex A]) are a vital aspect of a safe CAV system, as they are used by each vehicle to inform surrounding vehicles of their current status. Each vehicle needs to trust that the values contained within a CAM are timely and accurate. If this is not the case then autonomous vehicles could make incorrect and even unsafe decisions. Note that we focus here on CAMs generated by On-Board Units (OBU) in vehicles rather than Road Side Units (RSU) as the OBU algorithm for CAM generation is more complex and thus more interesting from a formal methods perspective. In terms of the CAM protocol, we specialise the STRIDE threats:

Spoofing: attacker sends messages masquerading as another vehicle.
Tampering: attacker tampers with a message sent by another vehicle.
Repudiation: a vehicle can deny sending a message that it has actually sent.
Information Disclosure: vehicles only receive messages intended for them.
Denial of Service: messages are not sent within a reasonable time frame.
Escalation of Privilege: attacker can send more CAMs than permitted.

Message Type	Message Element	Requirement	Attack Surfaces	Threats
Vehicle information [20]	Vehicle Type	Originating Station (RSU), Vehicle length, Vehicle width	Data system, Planning system, Wireless Comms	S, I, D
	Position	Reference position		
	Lane Position	Current Lane Position	Sensing: Lidar, Radar, Camera, Ultrasonic Positioning system: GPS, A-GPS Wireless Comms	S, T, R, I, D
	Speed	Vehicle velocity		
	Acceleration	Longitudinal, Lateral, Vertical		
	Heading	Heading		
	Driving model	Acceleration control		
	GPS	Preceding vehicle GPS, Following vehicle GPS		
Traffic notification [1]	Warning	Emergency vehicle, crash, collision.	Controlling centre Infrastructures, Wireless Comms	D, E
	Indication	Speed limits, traffic light		

Table 1: This table contains our threat analysis of the CAM protocol. Here, 'Requirement' corresponds to the information that the vehicle must sense about itself in order to know the value of the corresponding 'Message Element' on the left. Furthermore, the 'Threats' correspond to those identified using STRIDE.

The threat modelling contained in Table 1 was carried out by examining each piece of information that could be sent via CAMs and considering which STRIDE threats an attacker might exploit. The information contained in Table 1 is based on the C-ITS standard messages elements, a summary of threats [5, 6, 12, 18, 22, 25]. We summarise the information contained in Table 1 as follows.

For CAM, there are two distinct kinds of 'Message Type'. In particular, 'Vehicle information' CAMs include the vehicle type and its state information (speed, position, GPS, etc.). Conversely, 'Traffic notification' CAMs provide emergency warnings and traffic indications. For each kind of CAM, 'Message Element' indicates the specific components that are included in the messages. For each message element, its corresponding 'Requirement' refers to the information that the vehicle must sense/have access to in order to populate the corresponding message element field. In cyber security, an 'Attack Surface' is the region of the system that an adversary can exploit to attack the system. Finally, the possible 'Threats' for CAM are modelled based on STRIDE.

For example, in order to include GPS information we require the GPS information of both the leader and following vehicles. Here, the attack surface is the positioning system (GPS/A-GPS) and possible threats are Repudiation [25] or Spoofing [5].

The threat analysis contained in Table 1 has identified potential points of attack and we consider them in more detail in the next subsection. Note that, in the remainder of this paper, we focus on 'Vehicle information' CAMs but we have included 'Traffic notification' messages in Table 1 for illustrative purposes. These Decentralised Environment Notification Messages (DENM) are generated using a different protocol which we are not focusing on in this work.

3.2 Considering the Threats

We have identified a number of threats in Table 1 and, as part of the threat analysis process, we examine them in more detail here which allows us to identify which are the most likely to occur/cause the most damage.

In Table 1, we have identified Tampering as a threat for some of the state information contained in CAMs. However, in practice, Tampering is prevented via digital signatures and certificates, the verification of which is beyond the scope of this paper but details can be found in [16]. In particular, the CA Basic Service, which is responsible for operating the CAM protocol, interfaces with the SF-SAP security entity as described in [1, §5.1 & §6.2.2] which provides access to security services for CAM such as digital signing and certificates. Here, certificates are used to indicate the holder's privileges for sending CAMs. In this way, incoming CAMs are accepted if the sender's certificate is valid and is consistent with their privileges.

As CAMs are intended for all who receive them, we do not analyse Information Disclosure properties. Escalation of Privilege attacks could enable an attacker to send more messages than allowed, but in general all vehicles have the same level of authority so we do not consider this attack here.

Based on our analysis, we conclude that Spoofing, Denial of Service and Repudiation threats are the most relevant/important threats pertaining to this case study. To our knowledge, no formal, mathematical definition of the STRIDE properties exists since they are to be specialised for a given system. However, if we wish to include these in our formal verification of CAM then we must more closely consider those properties that we are interested in (Spoofing, Denial of Service and Repudiation). We explore these in more detail as follows:

Spoofing: an attacker pretends to be another vehicle and sends false information about that vehicle (e.g. speed) in CAMs. This could potentially cause vehicles to collide and we analyse this using Promela/SPIN in Section 4 by modelling an attacker of the system.

Denial of Service: a compromised vehicle does not send CAMs within a reasonable amount of time. If a vehicle sends too many CAMs then the network becomes overloaded. Conversely, if a vehicle does not send CAMs frequently enough then the most recent CAMs sent may be deemed out of date and thus ignored. In particular, a replay attack could occur where an attacker or a compromised vehicle resends CAMs that have already been sent causing a network overload. If suitable measures are not put in place to ensure that the time that the message was sent was not too far in the past then vehicles may react to an out of date message. We address this using Dafny in Section 5 by verifying an availability property of the algorithm for sending CAMs.

Repudiation: we can reduce the possibility of a vehicle denying that it has sent a CAM by requiring that CAMs are stored in a sequence and not providing functionality to remove CAMs from this sequence. Another repudiation related attack could result in an attacker or compromised vehicle claiming to have sent a CAM when in fact it has not sent one. In this case, vehicles could

potentially forward CAMs to other vehicles. This is a particular condition that is prohibited in the documentation [1] and our Dafny implementation of the algorithm for receiving CAMs in Section 5 considers this.

These are the threats that, based on our threat analysis, we consider to be the most relevant/likely with respect to the CAM protocol and we use these to guide our formal verification effort[5].

4 Model-Checking with Promela/SPIN

In this case study, it is easy to see that safety and security are inextricably linked. For CAV systems, the most important safety property to consider is that collisions should be avoided at all costs. Therefore, an attacker of the system who is attempting to cause harm will likely target security vulnerabilities that have the potential to violate this safety property. A key aspect of the threat analysis process is the identification of a suitable attacker. To this end, we recognise that there may be malicious vehicles on the road that are attempting to cause vehicles to collide, perhaps a disgruntled taxi driver who is unemployed due to the adoption of autonomous vehicles. Such a collision could be caused by spoofing the CAMs sent between vehicles. Our analysis of spoofing and how it can impact the safety of the CAV system is captured here in a SPIN analysis of a simplified scenario involving CAMs between vehicles in a platoon/convoy.

4.1 Basic Scenario: Safety

We investigated message passing between multiple vehicles by applying SPIN to an abstracted Promela model for sending and receiving CAMs. Fig. 2 contains three vehicles travelling in a platoon/convoy: one leader; one middle; and one tail vehicle. The leader and tail send CAMs to the middle vehicle, and it follows a simple protocol.

- If no CAMs are received then it continues unchanged.
- If it receives exactly one CAM then sets its own speed to half the speed in the CAM.[6]
- If it receives two CAMs then it sets its own speed to be the average of the two speeds (rounded down).

We used the following default conditions to analyse this Promela model with SPIN: (1) the leader chooses a random discrete speed 10, 20, 30, 40, 50, 60 or 70 at each time step,(2) the tail similarly chooses a random discrete speed at each time step, and (3) we ran the system for 100 time steps with a round-robin interleaving concurrency between vehicles.

The simple safety property that we verified is that the speed of the middle vehicle is never *much* different to the speed of the leader or of the tail, for more

[5] Artefacts available at: `https://github.com/mariefarrell/CAMVerification.git`
[6] This only occurs at initialisation when the speed of the other vehicle is 0.

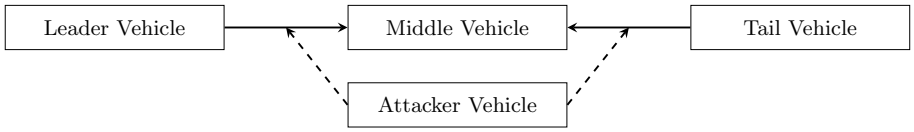

Fig. 2: Our three vehicle model where CAMs are sent from the leader and tail vehicles to the middle vehicle. The attacker executes a Spoofing attack.

than one step. Here, *much different* means a difference of more than '51' in speed. We formalise: it is always the case that, if the speed of the middle vehicle is *much different* then it will not be in the next state. We write this in temporal logic as:

$$\Box(big\_speed\_difference \Rightarrow \bigcirc \neg big\_speed\_difference)$$

where '$\Box$' and '$\bigcirc$' are LTL's [23] "always" and "in the next state"operators, respectively. Although we have only written $\bigcirc$ here for ease of presentation, in the actual implementation the property is that the difference has been corrected after twelve next steps – this is because SPIN treats next as the next instruction execution, which includes print states used for understandability and debugging purposes not as the next tick of the internal clock. In SPIN, we negate this property and so the "never claim" (or safety property [4]) that we implement is

$$\Diamond(big\_speed\_difference \wedge \bigcirc big\_speed\_difference)$$

where '$\Diamond$' is LTL's [23] "eventually" operator. Here, *big_speed_difference* is `true` when any of the following inequalities hold

- middle vehicle speed > (leader vehicle speed + 51)
- middle vehicle speed < (leader vehicle speed - 51)
- middle vehicle speed > (tail vehicle speed + 51)
- middle vehicle speed < (tail vehicle speed - 51)

We have successfully verified that this safety property holds of our model using the SPIN model-checker. Next, we use this model to investigate how a Spoofing attack could lead to an unsafe scenario for the vehicle platoon.

4.2 Investigating Spoofing

Inspired by the threat analysis contained in Table 1, we have modelled a Spoofing attack in Promela for the above scenario. In order to analyse this kind of threat, we add a process to our Promela model to describe an attacker of the system. The behaviour of this attacker (as illustrated in Fig. 2 and Fig. 3) is as follows:

- At one point in the execution trace the attacker may insert two speed messages into the channel between the leader and middle vehicle stating that the leader's speed is 10, 20, 30, ... or 70 (lines 6–10 of Fig. 3).

```
0  proctype attacker(chan l_in, t_in){   /* attacker */
1    printf("attacker: starting\n");
2    bool head = 0;
3    bool tl = 0;
4    A:    (clock > 10); /* wait until under way */
5      if
6        :: (head = 0) -> printf("attacker: inserting vspeed of 10\n");
7             l_in!10; l_in!10; head = 1; goto A;
8                  ...
9        :: (head = 0) -> printf("attacker: inserting vspeed of 70\n");
10            l_in!70; l_in!70; head = 1; goto A;
11       :: (tl = 0) -> printf("attacker: inserting tspeed of 10\n");
12            t_in!10; t_in!10; tl = 1; goto A;
13                 ...
14       :: (tl = 0) -> printf("attacker: inserting tspeed of 70\n");
15            t_in!70; t_in!70; tl = 1; goto A;
16       :: (clock ≤ 100) -> goto A;
17       :: (clock > 100) -> goto FIN;
18     fi;
19   FIN:  printf("attacker: finishing\n")
20 }
```

Fig. 3: Promela model of the attacker.

– At one point in the execution trace the attacker may insert two speed messages into the channel between the tail and the middle vehicle stating that the leader's speed is 10, 20, 30, ... or 70 (lines 11–15 of Fig. 3).

We note that in each of the above cases, both the speed, whether to insert a message and the time that the message is inserted are chosen at random. Running SPIN with this attacker model and the initial model described above, we can see that our $\Box(\mathit{big\_speed\_difference} \Rightarrow \bigcirc\neg \mathit{big\_speed\_difference})$ property has been violated. It is important to note that this is a deliberately simple example but scales up to more complex versions of such Spoofing attacks.

4.3 Discussion

In Section 3, cyber security threat analysis focused the whole vehicle security area to scenarios, such as the one illustrated in Fig. 2, that were identified as high risk. In particular, this threat that was identified following STRIDE and analysed using Promela/SPIN could potentially lead to an unsafe scenario causing vehicles to collide. Note that whilst the above example deals with modelling and verification of aspects of a platoon/convoy, where a group of vehicles drive together with a leader, this could be generalised to messages between autonomous vehicles driving without a platoon.

The evidence that we have collected above illustrates how a spoofing attack on this system can negatively impact its safe operation. We have focused on speed, but using model-checking, we can explore whether spoofing of other attributes, as identified in Table 1, can impact safety. These results can help to strengthen the argument as to why mitigations should be put in place against specific threats. Our simple attacker model has allowed us to identify that a

spoofing attack is indeed possible for this scenario. In practice, mitigations would be put in place against this kind of attack. Then, our simple model would be refined to add these mitigations and would then undergo further verification.

In particular, those implementing the CAM protocol should consider the possibility that malicious vehicles may join the platoon with the sole aim of causing collisions. Based on our formal Promela model, runtime monitors could be synthesised to monitor the CAMs being sent between vehicles so that this spoofing attack could be recognised and prevent it from causing harm.

Our Promela model that describes an attacker and three vehicles is only one scenario that could occur, particularly as there may be many more vehicles in a real world scenario. To our knowledge, there is no systematic way of identifying all possible models of the system that include a spoofing attack. However, we can systematically work through the attributes that have been identified in Table 1 as likely to be vulnerable to spoofing to examine how spoofing attacks can influence safe system behaviour.

5 Deductive Verification with Dafny

In this section, we construct and verify a CAM send and receive implementation using Dafny. Our Dafny implementation of CAM contains two basic methods; `sendCAM` (Fig. 4) and `receiveCAM` (Fig. 5). We have formalised the specification of CAM using the available documentation [1, §6.1.3] and followed its nomenclature. As is to be expected when following the associated documentation, quite some time was taken when constructing the formal specification from the informal, English-language description of the CAM protocol contained in [1].

Our verification of `sendCAM` and `receiveCAM` in Dafny focuses on the Denial of Service and Repudiation security threats, this time at the algorithmic level. In our implementation we have simplified the structure of CAMs from the ASN.1 encoding to focus on the semantic contents of the message as follows:

```
CAM(id:int, time:int, heading:int, speed:int, position:int)
```

Here, `id` refers to the vehicle that is sending the CAM and `time` is the timestamp at which the CAM was sent. These attributes are required by the documentation [1]. As mentioned earlier, CAMs are sent periodically, or when any of the status information (e.g. speed) contained in the message has changed since the last message was sent.

5.1 Sending CAMs

Fig. 4 contains the verified Dafny code corresponding to the `sendCAM` algorithm which is responsible for generation and transmission of CAMs. We describe the key components of the Dafny algorithm in Fig. 4 as follows:

Lines 0–3: Since CAMs should be sent periodically within time bounds specified by the CA Basic Service, this method takes two variables as input. `T_CheckCamGen` describes how often to check if another CAM should be sent

```
0  method sendCAM(T_CheckCamGen: int, T_GenCam_DCC: int)
1  returns (msgs: seq<CAM>, now: int)
2  requires 0 < T_CheckCamGen ≤ T_GenCamMin;
3  requires T_GenCamMin ≤ T_GenCam_DCC ≤ T_GenCamMax;
4  ensures T_GenCam_DCC * |msgs| ≤ now ≤ T_GenCamMax * |msgs|;
5  ensures |msgs| ≥ 2 ⟹ ∀ i: int • 1 ≤ i < |msgs| ⟹
6      T_GenCam_DCC ≤ (msgs[i].time - msgs[i-1].time) ≤ T_GenCamMax;
7  ensures |msgs| = MaxMsgs;
8  {
9    var T_GenCam, T_GenCamNext, j := T_GenCamMax, T_GenCamMax, GetId();
10   var N_GenCam, trigger_two_count := N_GenCamDefault, 0;
11   msgs, now := [], 0;
12   var LastBroadcast, PrevLastBroadcast,prevsent := now, now, msgs;
13   var heading, speed, pos := GetHeading(), GetSpeed(), GetPosition();
14   var prevheading, prevspeed, prevpos, statechanged := -1, -1, -1, false;
15
16   while (|msgs| < MaxMsgs)
17   decreases MaxMsgs - |msgs|;
18   invariant 0 ≤ |msgs| ≤ MaxMsgs ∧ 0 < N_GenCam ≤ N_GenCamMax;
19   invariant T_GenCamMin ≤ T_GenCamNext ≤ T_GenCamMax;
20   invariant T_GenCamMin ≤ T_GenCam ≤ T_GenCamMax;
21   invariant 0 ≤ PrevLastBroadcast ≤ now ∧ now = LastBroadcast;
22   invariant now - T_GenCamMax ≤ PrevLastBroadcast ≤ LastBroadcast;
23   invariant |msgs| ≥ 1 ⟹ msgs[|msgs|-1].time = LastBroadcast;
24   invariant |msgs| ≥ 2 ⟹ msgs[|msgs|-2].time = PrevLastBroadcast;
25   invariant now > 0 ⟹ T_GenCam_DCC ≤ LastBroadcast - PrevLastBroadcast ≤
         T_GenCamMax;
26   invariant now > 0 ⟹ CAM(j,now,heading,speed,pos) in msgs;
27   invariant now > 0 ⟹ |prevsent| + 1 = |msgs|;
28   invariant |msgs| ≥ 2 ⟹ ∀ i: int • 1 ≤ i < |msgs| ⟹
29     T_GenCam_DCC ≤ (msgs[i].time - msgs[i-1].time) ≤ T_GenCamMax;
30   invariant T_GenCamMin * |msgs| ≤ T_GenCam_DCC * |msgs| ≤ now;
31   invariant now > 0 ⟹ now ≤ T_GenCamMax * |msgs|;
32   {
33     prevsent, PrevLastBroadcast := msgs, LastBroadcast;
34     T_GenCam, statechanged := T_GenCamNext, false;
35     now := now + T_GenCam_DCC;
36
37     while (true)
38     decreases LastBroadcast + T_GenCam - now;
39     invariant now - LastBroadcast ≤ max(T_GenCam_DCC, T_GenCam);
40     {
41       heading, speed, pos := GetHeading(), GetSpeed(), GetPosition();
42       statechanged := abs(heading - prevheading) ≥ headingthreshold ∨
43                       abs(speed - prevspeed) ≥ speedthreshold ∨
44                       abs(pos - prevpos) ≥ posthreshold.Floor;
45
46       if (statechanged ∨ now - LastBroadcast ≥ T_GenCam) { break; }
47       else { now := now + T_CheckCamGen; }
48     }
49     msgs := msgs + [CAM(j,now,heading,speed,pos)];
50
51     if (statechanged) {
52       T_GenCamNext, trigger_two_count := now - LastBroadcast, 0;
53     }
54     else if (now - LastBroadcast ≥ T_GenCam){
55       trigger_two_count := trigger_two_count + 1;
56       if (trigger_two_count = N_GenCam) { T_GenCamNext := T_GenCamMax; }
57     }
58     LastBroadcast := now;
59     prevheading, prevspeed, prevpos := heading, speed, pos;
60   }
61   return msgs, now;
62 }
```

Fig. 4: Dafny implementation of the `sendCAM` algorithm. We have specified the Denial of Service property as a postcondition on lines 4–5.

and `T_GenCam_DCC` which describes the minimum time interval between two consecutive CAM generations. It returns a sequence of CAMs that have been sent, denoted by `msgs`, and the current time given by the variable, `now`. The preconditions, indicated by the `requires` keyword on lines 2 and 3, provide constraints on these variables. In particular, `T_GenCam_DCC` is required to be between `T_GenCamMin` (100ms) and `T_GenCamMax` (1000ms) [1, §6.1.3].

Lines 4–7: The postconditions on lines 4–7, indicated by the `ensures` keyword, specify that the expected number of CAMs have been sent and that these messages were sent within the required time bounds. This corresponds to the Denial of Service threat by ensuring that messages are sent on time and arrive within specified time bounds. In particular, line 4 provides constraints on the value of the current time. This is necessary because Dafny does not support real-time systems so we had to manually keep track of time. The postcondition on line 5 specifies that the interval between any two consecutive CAMs is between `T_GenCam_DCC` and `T_GenCamMax` as described in [1]. For the purpose of discretising the system, the postcondition on line 7 ensures that the maximum number (`MaxMsgs := 100`) of CAMs are sent[7].

Lines 8–14: Here, we initialise the relevant local variables. In particular, we set `msgs` to the empty sequence and `now` to 0 (line 11). Some of these variables are specified in the CAM documentation but others are not and we include them for implementation purposes. In particular, `T_GenCam` as defined on line 9 represents the current upper limit of the CAM generation interval, by default this is equal to `T_GenCam_Max` [1, §6.1.3]. We also assume the existence of verified helper functions for `GetHeading()`, `GetSpeed()` and `GetPosition()` as used on line 13.

Lines 16–17: The method loops until `MaxMsgs` number of CAMs have been sent. In order to prove termination of the loop, we specify the loop variant as indicated by the `decreases` keyword on line 17.

Lines 18–25: We specify these loop invariants to ensure that the relevant variables stay within the allowable bounds during loop execution. In particular, the invariant on line 18 relates to the postcondition on line 7 by specifying that the number of CAMs sent so far is less than or equal to `MaxMsgs`.

Lines 26–27 These invariants ensure that once time has begun then at least one CAM has been sent.

Lines 28–31: These invariants relate to the postcondition on lines 4–6 and thus relate to the availability property described earlier.

Lines 32–36: During each loop iteration we update the appropriate variables. Note that we increment the current time, `now`, by `T_GenCam_DCC` to allow time to advance until the earliest time that the next CAM can be sent.

Lines 37–50: This inner loop checks if any state information has changed and updates the `statechanged` variable accordingly. Note that the variables `headingthreshold`, `speedthreshold` and `posthreshold` are global and their values are controlled by the CA Basic Service [1] as described in Section 2. If the autonomous vehicle's state has changed or it is time to send

[7] We chose 100 as a value but we could easily have chosen some other value.

```
0  method receiveCAM(fromid: int, cams: seq<CAM>, now: int) returns (brake: bool)
1  requires 0 ≤ fromid < |cams|;
2  requires fromid = cams[fromid].id;
3  ensures !(now - cams[fromid].time > T_GenCamMax)
4          ∧ Sign(Magnitude(cams[fromid].heading)) = Sign(Magnitude(GetHeading
               (now)))
5          ∧ GetSpeed(now) - cams[fromid].speed < 0 ⟹ brake;
6  ensures now - cams[fromid].time > T_GenCamMax ⟹ !brake;
7  {
8    var speeddiff := 0;
9
10   if (now - cams[fromid].time > T_GenCamMax){
11       brake := false;
12     }
13   else if(Sign(Magnitude(cams[fromid].heading)) = Sign(Magnitude(GetHeading
         (now))))
14   {
15     speeddiff := GetSpeed(now) - cams[fromid].speed;
16
17     if (speeddiff < 0){
18      brake:=true;
19     }
20   }
21 }
```

Fig. 5: Dafny implementation of the `receiveCAM` algorithm.

another CAM then we break from this inner loop. Otherwise, nothing has changed so we keep looping to allow time to advance until either the state has changed or sufficient time has passed since the last CAM was sent. Once we have exited this inner loop then a CAM is sent.

Lines 51–57: Based on the reason that the CAM was sent, i.e. whether the state changed or it was simply time to send a CAM, this if-else statement updates the relevant variables as described in [1, §6.1.3].

Lines 58–62: Finally, we update and return the appropriate variables.

In this way, the Dafny algorithm illustrated in Fig. 4 is verified with respect to the STRIDE Denial of Service threat (or Availability property). We also verified other correctness properties that were derived from the documentation [1]. As mentioned above, it was necessary to discretise some components of the specification. In fact, discretising the continuous features of autonomous systems is a common challenge for formal methods [19]. As already discussed, the CA Basic Service also facilitates the receiving of CAMs and we describe our Dafny implementation of the receive method in the next subsection.

5.2 Receiving CAMs

Fig. 5 contains our Dafny implementation of the `receiveCAM` algorithm which takes as input the id of the vehicle sending the CAM (`fromid`), the sequence of CAMs that have been sent (`cams`) and the current time (`now`).

We have used this to verify a simple Non-Repudiation property as specified by the preconditions on lines 1–2. We assume that CAMs are uploaded to a sequence that can then be accessed by the other vehicles nearby. The latest CAM

for each car is stored at a position in the sequence that matches its vehicle id number. We express the non-repudiation property by requiring that the received CAM did indeed come from a vehicle with a valid id and that the vehicle claiming to have sent the CAM did actually send one. §6.1.1 of [1] specifies that any received CAMs should not be forwarded to other vehicles in the intelligent transport system and our preconditions capture this by requiring that the sender did actually send a CAM.

Of course, CAMs are used by the receiving vehicles to modify their state with respect to the information that they receive. For example, if a leader vehicle decreases their speed then a vehicle that is travelling behind it should also reduce their speed, provided that they are travelling in the same direction. To this end, our `receiveCAM` implementation in Fig. 5 also describes when the vehicle should brake. We specify this safety property as a postcondition on lines 3–6. In particular, if the current vehicle and the one that sent the CAM are travelling in the same direction and the current vehicle has a greater speed than the one in front, then the brake should be engaged.

Without the security property on line 1–2, the safety property can still be verified. However, if the security property is violated and an attacker is sending a false message to the receiving vehicle, potentially that the leader has not slowed down when they have, then there could be a collision even though the safety property on lines 3–6 is preserved. This illustrates the importance of considering security properties alongside safety for these complex and connected systems where security violations can impact safety. In reality, the braking mechanism would be more complex than simply toggling a boolean flag as we have done above, however, the same basic properties apply.

5.3 Discussion

One advantage of Dafny for this case study is that we were able to run tests in Visual Studio to complement the formal verification results presented above. Crucially, Dafny is relatively easy to communicate to security practitioners since it more closely resembles the implementation language than other formal methods such as Promela/SPIN (Section 4). However, since it is not a language that can be used for the final implementation, some discrepancies may exist between our implementation and the one used in the fully implemented system. In particular, a more realistic version of this algorithm would keep track of whether the receiving vehicle are getting closer to the vehicle in front or not rather than just focusing on the speed part of the CAM and this could be seen as a refinement of our original model.

Note that the Denial of Service property that we have verified in the `sendCAM` method only applies if the attacker is trying to flood the network with CAMs, and does not address the scenario when they might use other kinds of messages. However, our approach could be extended to other message types in vehicle-to-vehicle communications, such as DENM [26].

An open question in software verification is in ensuring that the verified models faithfully capture what happens in the fully implemented systems. This

"reality gap" is difficult to traverse and will almost always exist when building abstract models of program behaviour [8, 19]. Since all real world implementations of CAM should comply to the specification outlined in [1], we chose it as our starting point for modelling this protocol. We could potentially run the Dafny implementation alongside a real world implementation and check that they exhibit the same behaviour but this was out of the scope of this work.

6 Conclusions and Future Work

This paper presents a case study showing how cyber security threat analysis techniques can be used to guide formal methods practitioners in verifying security properties, particularly as they may impact safety. Previously, we discussed the need for the use of integrated formal methods in the robotics domain and the example that we present here is no different [8].

We carried out STRIDE threat analysis of the CAM protocol for sending and receiving messages between autonomous vehicles. This resulted in the identification of spoofing, denial of service and repudiation as attacks that may occur. We modelled spoofing by specifying the behaviour of an attacker in our Promela model. Denial of service was considered via an availability property in the Dafny implementation of the algorithm for sending CAMs. Finally, repudiation was addressed as a property to be verified of the Dafny algorithm for receiving CAMs.

By modelling the system at different levels of abstraction; system-level in Promela/SPIN and algorithm-level in Dafny, we were able to investigate and to verify properties related to STRIDE threat analysis. In particular, model-checking with Promela/SPIN is useful for examining high-level temporal properties. Conversely, the use of theorem proving with Dafny allowed us to examine properties of an implementation of the CAM protocol. Our use of distinct tools allowed us to examine different properties of the CAM protocol at different levels of abstraction. Future analysis of CAM with various tools will likely provide a better understanding of which STRIDE properties should be checked using different kinds of formal methods.

An important aspect here is that, although it could be useful, the individual formal analyses do not need to be combined as in holistic/compositional formal approaches [2,13,21]. Instead, formal methods are used to focus security analysis on to specific areas/scenarios highlighted by informal cyber security analysis as being of "high risk". However, an interesting avenue of future work might involve proving that the independent formal models do, in fact, capture the same system.

This work is a first step toward a detailed methodology of how STRIDE properties should be treated in formal verification. Therefore, our future work aims to define a more general methodology for combining threat analysis techniques and formal methods. Of course, our use of Promela/SPIN and Dafny has been motivated by our familiarity with these tools and it is certainly the case that other formal methods may have been a better choice for our study. We intend to investigate this further in future work.

References

1. Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service. Standard Draft ETSI EN 302 637-2, European Telecommunications Standards Institute, Nov. 2018. V1.4.0 (2018-08).
2. R.-J. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
4. M. Ben-Ari. *Principles of the Spin model checker*. Springer, 2008.
5. S. Bittl, A. A. Gonzalez, M. Myrtus, H. Beckmann, S. Sailer, and B. Eissfeller. Emerging attacks on vanet security based on gps time spoofing. In *IEEE Conference on Communications and Network Security*, pages 344–352. IEEE, 2015.
6. J. Choi and S.-i. Jin. Security threats in connected car environment and proposal of in-vehicle infotainment-based access control mechanism. In *Advanced Multimedia and Ubiquitous Engineering*, volume 518 of *LNEE*, pages 383–388. Springer, 2018.
7. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
8. M. Farrell, M. Luckcuck, and M. Fisher. Robotics and Integrated Formal Methods: Necessity meets Opportunity. In *Integrated Formal Methods*, volume 11023 of *LNCS*, pages 161–171. Springer, 2018.
9. M. Fisher. *An Introduction to Practical Formal Methods Using Temporal Logic*. Wiley, 2011.
10. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
11. L. Huang and E.-Y. Kang. Formal verification of safety & security related timing constraints for a cooperative automotive system. In *Fundamental Approaches to Software Engineering*, volume 11424 of *LNCS*, pages 210–227. Springer, 2019.
12. M. Jagielski, N. Jones, C.-W. Lin, C. Nita-Rotaru, and S. Shiraishi. Threat detection for collaborative adaptive cruise control in connected cars. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 184–189. ACM, 2018.
13. C. B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.
14. M. Kamali, S. Linker, and M. Fisher. Modular verification of vehicle platooning with respect to decisions, space and time. In *International Workshop on Formal Techniques for Safety-Critical Systems*, volume 1008 of *CCIS*, pages 18–36. Springer, 2018.
15. L. Kohnfelder and P. Garg. The Threats to Our Products. `adam.shostack.org/microsoft/The-Threats-To-Our-Products.docx`, Apr. 1999. Accessed: 2018-12-10.
16. B. Langenstein, R. Vogt, and M. Ullmann. The use of formal methods for trusted digital signature devices. In *Florida Artificial Intelligence Research Society*, pages 336–340. AAAI Press, 2000.
17. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming Artificial Intelligence and Reasoning*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

18. J. Liu, C. Yan, and W. Xu. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicles. *DEFCON24 (Aug. 2016); http://bit.ly/2EQNOLs*, 2016.
19. M. Luckcuck, M. Farrell, L. Dennis, C. Dixon, and M. Fisher. Formal specification and verification of autonomous robotic systems: A survey. *accepted, ACM Computing Surveys*, 2019.
20. A. C. Michele Rondinone. Deliverable (d) no: 5.1 definition of v2x message sets. report, Universidad Miguel Hernandez, Aug. 2018. V1.0 27/08/2018.
21. C. Morgan, K. Robinson, and P. Gardiner. *On the Refinement Calculus*. Springer, 1988.
22. J. Petit, B. Stottelaar, M. Feiri, and F. Kargl. Remote attacks on automated vehicles sensors: Experiments on camera and lidar. *Black Hat Europe*, 11:2015, 2015.
23. A. Pnueli. The Temporal Logic of Programs. In *18th Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.
24. R. S. Ross. Guide for Conducting Risk Assessments. Technical report, National Institute of Standards and Technology, Sept. 2012. SP 800-30 Rev. 1.
25. A. Ruddle, D. Ward, B. Weyl, S. Idrees, Y. Roudier, M. Friedewald, T. Leimbach, A. Fuchs, S. Gürgens, O. Henniger, et al. Security requirements for automotive on-board networks based on dark-side scenarios. *EVITA Deliverable D*, 2:3, 2009.
26. J. Santa, F. Pereñíguez, A. Moragón, and A. F. Skarmeta. Vehicle-to-infrastructure messaging proposal based on cam/denm specifications. In *Wireless Days (WD), IFIP*, pages 1–7. IEEE, 2013.
27. S. Schneider. Formal analysis of a non-repudiation protocol. In *Computer Security Foundations Workshop*, pages 54–65. IEEE, 1998.
28. S. Schneider. Verifying authentication protocols in csp. *IEEE Transactions on Software Engineering*, 24(9):741–758, 1998.
29. S. Schneider and R. Delicata. Verifying security protocols: An application of csp. In *Communicating Sequential Processes. The First 25 Years*, volume 3525 of *LNCS*, pages 243–263. Springer, 2005.
30. C. Snook, T. S. Hoang, and M. Butler. Analysing security protocols using refinement in iUML-B. In *NASA Formal Methods Symposium*, volume 10227 of *LNCS*, pages 84–98. Springer, 2017.
31. W. Stallings, L. Brown, M. D. Bauer, and A. K. Bhattacharjee. *Computer security: principles and practice*. Pearson, 2012.
32. G. Vanspauwen and B. Jacobs. Verifying protocol implementations by augmenting existing cryptographic libraries with specifications. In *Software Engineering and Formal Methods*, volume 9276 of *LNCS*, pages 53–68. Springer, 2015.
33. J. Whitefield, L. Chen, F. Kargl, A. Paverd, S. Schneider, H. Treharne, and S. Wesemeyer. Formal analysis of v2x revocation protocols. In *Security and Trust Management*, volume 10547 of *LNCS*, pages 147–163. Springer, 2017.